VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY

FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZENÍ STÍNŮ Z VŠESMĚROVÝCH SVĚTELNÝCH ZDROJŮ

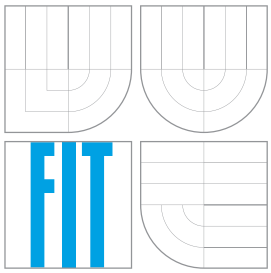
BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE
AUTHOR

TOMÁŠ MIKULICA

BRNO 2013

VYSOKÉ UČENÍ TECHNICKÉ V BRNĚ
BRNO UNIVERSITY OF TECHNOLOGY



FAKULTA INFORMAČNÍCH TECHNOLOGIÍ
ÚSTAV POČÍTAČOVÉ GRAFIKY A MULTIMÉDIÍ

FACULTY OF INFORMATION TECHNOLOGY
DEPARTMENT OF COMPUTER GRAPHICS AND MULTIMEDIA

ZOBRAZENÍ STÍNŮ Z VŠESMĚROVÝCH SVĚTELNÝCH ZDROJŮ

SHADOW RENDERING FROM OMNIDIRECTIONAL LIGHT SOURCES

BAKALÁŘSKÁ PRÁCE
BACHELOR'S THESIS

AUTOR PRÁCE — TOMÁŠ MIKULICA
AUTHOR

VEDOUCÍ PRÁCE — Ing. JAN NAVRÁTIL
SUPERVISOR

BRNO 2013

Abstrakt

Tato práce pojednává o možnostech zobrazení stínů z všesměrových světelných zdrojů za použití knihovny OpenGL. Zejména se zaměřuje na algoritmy Cube map shadow mapping a Dual paraboloid shadow mapping. Dále jsou v práci obsaženy výsledky srovnání implementací těchto algoritmů z hlediska časové náročnosti vytvoření stínové mapy a také z hlediska vizuální kvality stínů.

Abstract

This work discusses the possibilities for shadow rendering from omnidirectional light sources using OpenGL library. In this work the Cube map shadow mapping and the Dual paraboloid shadow mapping algorithms are described. Further more, this work contains the results of a comparison of these two methods in a way of time required for shadow map creation and a comparison of visual quality of resulting shadows.

Klíčová slova

Stíny, všesměrové světelné zdroje, parabolická projekce, OpenGL, Shadow Mapping, GLSL, Cube mapa

Keywords

Shadows, omnidirectional lighting sources, dual paraboloid projection, OpenGL, Shadow Mapping, GLSL, cube map

Citace

Tomáš Mikulica: Zobrazení stínů z všesměrových světelných zdrojů, bakalářská práce, Brno, FIT VUT v Brně, 2013

Zobrazení stínů z všesměrových světelných zdrojů

Prohlášení

Prohlašuji, že jsem tuto bakalářskou práci vypracoval samostatně pod vedením pana Ing. Jana Navrátila

.....
Tomáš Mikulica
12. května 2013

Poděkování

Chtěl bych poděkovat hlavně vedoucímu práce Ing. Janu Navrátilovi za ochotu a za konstruktivní připomínky, které mě pomohly dovést tuto práci do konce.

Obsah

1 Úvod **2**

2 Stínové algortimy **3**
2.1 Shadow volumes 3
2.2 Shadow Mapping 4
2.3 Cube Map Shadow Mapping 6
2.4 Dual Paraboloid Shadow Mapping 8

3 Světelné zdroje **11**
3.1 Směrové zdroje 11
3.2 Všesměrové zdroje 11
3.3 Reflektor 12
3.4 Plošné zdroje 12
3.5 Ostré a měkké stíny 12

4 Demonstrační aplikace **14**
4.1 Návrh 14
4.2 Implementace 14

5 Srovnání metod **20**
5.1 Čas vykreslení 20
5.2 Kvalitativní porovnání 22

6 Závěr **26**

A Obsah CD **29**

Kapitola 1

Úvod

Jedním z cílů v počítačové grafice je zobrazit výslednou scénu tak, aby vypadala, co nejvěrohodněji a nejreálněji. Důležitou roli hraje i zobrazení stínů. Jelikož v reálném světě jsou světlo a stín neodmyslitelně k sobě vázány a nikdy nenastane situace, že objekt, který je osvětlen, tak nevrhá stín. Stíny nám pomáhájí při prostorovém vnímání. Ať už jde o vnímání relativní pozice a velikosti objektu nebo o směr odkud dopadá světlo. Mimo to nám napomáhájí vnímat strukturu jako jsou například různé nerovnosti povrchu. A lze je považovat za tmavou oblast, kam nedopadá světlo.

V počítačové grafice je zobrazení stínů nelehký úkol. Algoritmy pro výpočet stínů jsou již dlouhá léta předmětem výzkumu. Dokonce i v dnešní době je zobrazení stínů v reálném čase výpočetně náročné. A je nutné hledat kompromis mezi kvalitou a rychlostí. Globální zobrazovací metody jako například sledování paprsku poskytují věrohodné zobrazení scény včetně stínů. Avšak jsou výpočetně náročné. V dnešní době se nejčastěji používají algoritmy využívající hloubkový buffer a to především pro jejich rychlost a hardwarovou akceleraci.

V této práci se budeme zabývat hlavně algoritmy *Cube map shadow mapping* a *Dual paraboloid shadow mapping* a jejich implementací za použití knihovny *OpenGL* a jazyka pro programování shaderů *GLSL*. U obou těchto algoritmů se podíváme na srovnání jejich rychlosti vykreslování a také na výslednou kvalitu stínů. U algoritmu *Cube map shadow mapping* se zaměříme na jednu optimalizační metodu, která zvyšuje rychlost vykreslení této metody.

Kapitola 2 popisuje vybrané stínové algoritmy, které zvládají zobrazení stínů z všesměrových světelných zdrojů, a dále popisuje jejich výhody a nevýhody. V další kapitole 3 si představíme základní typy světelných zdrojů a jejich vliv na zobrazení stínů. Problematika návrhu a zejména implementace aplikace je rozebrána v kapitole 4. V předposlední kapitole 5 se zaměříme na výkonostní a kvalitativní porovnání implementovaných algoritmů. Na závěr si zrhneme dosažené výsledky a rozebereme další možný vývoj.

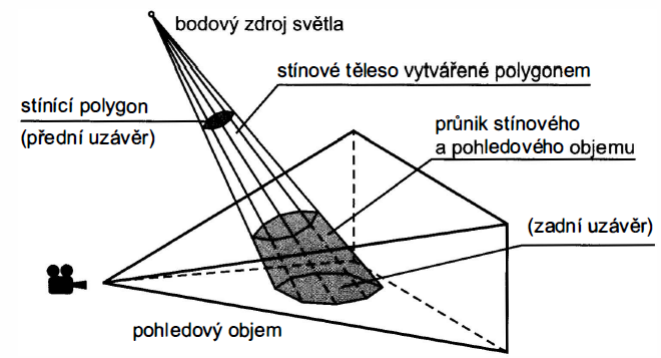
Kapitola 2

Stínové algortimy

Stínové algoritmy nejsou běžnou součástí vykreslovacího řetězce knihovny *OpenGL*. Neexistuje žádná funkce, která vykreslí stíny. Nicméně je možné tyto algortimy implementovat za použití prostředků z knihovny *OpenGL*. V této kapitole si představíme základní algoritmy, zejména ty, které dokáží zobrazovat stíny ze všesměrových světelných zdrojů.

2.1 Shadow volumes

Tento algoritmus představil v roce 1977 *Franklin C. Crow*[6]. A již ve své základní podobě pracuje s bodovými světlenými zdroji. Dále je nutné podotknout, že pracuje s polygony – nutná znalost geometrie scény. Principem této metody je vytvoření stínového tělesa – *shadow volume* pro každý ze stínících objektů. Stínové těleso určuje tu část scény, která je zastíněna. Po vytvoření stínových těles je nutné pro každé těleso ve scéně ověřit jeho polohu vůči stínovému tělesu. Podle výsledku tohoto testu se určí, zda je polygon osvětlen, zastíněn nebo částečně zastíněn. Pokud nastane poslední jmenovaný případ, je nutné rozdělit těleso v místě průniku se stínovým tělesem. A stejně jako algoritmy využívající hloubkový buffer, produkuje pouze ostré stíny. Nicméně v této práci se s algoritmem stínových těles nepracuje.



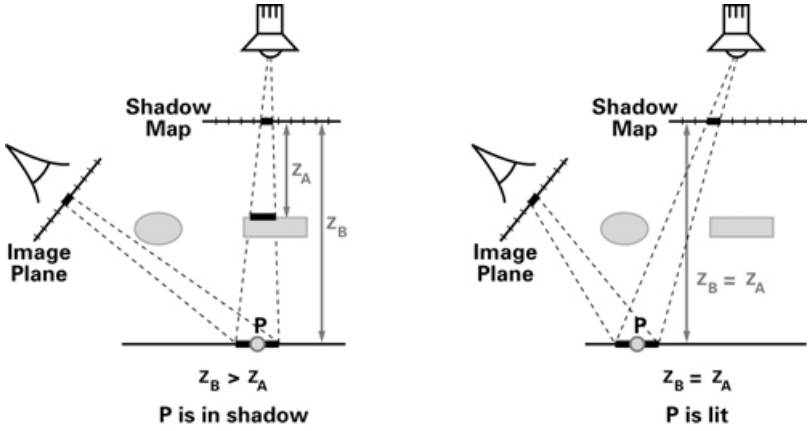
Obrázek 2.1: Stínové těleso [13].

2.2 Shadow Mapping

Shadow mapping je jedna z technik používaných pro zobrazení stínů ve scéně. Poprvé byla představena na konferenci *SIGGRAPH* v roce 1978 v článku *Casting curved shadows on curved surfaces*[15]. Tato technika ve své základní variantě potřebuje dva vykreslovací průchody scénou. Principem je vytvoření stínové mapy z pohledu zdroje světla a její následná aplikace na scénu. Stínová mapa musí pokrýt celou scénu. Z toho vyplívá, že je vhodná pouze pro směrové světelné zdroje.

Stínová mapa nebo také hloubková mapa je textura, kde ovšem není pro každý pixel uložena informace o barvě, ale informace o hloubce (vzdálenosti) pixelu. Tato textura se dále používá pro porovnání, zda je pixel ve scéně osvětlen nebo leží ve stínu. A představuje tu část scény, která je osvětlena od zdroje světla.

Výhodou této techniky je také fakt, že je nezávislá na složitosti scény. Dalším kladem je také hardwarová akcelerace. Nevýhodou je ovšem vliv velikosti (rozlišení) stínové mapy na výslednou kvalitu stínů a s tím spojený aliasing stínů. Dalším nepříjemným jevem je chybný tzv. *self shadowing* – chybné vrhání stínu na sebe sama, stínové akné.



Obrázek 2.2: Demonstrace porovnání hloubky pixelu a hloubky pixelu uložené ve stínové mapě [2].

2.2.1 Algoritmus

1. Vykresli scénu s vyplým osvětlením, stínováním a bez textur z pohledu světla a ulož její hloubku do textury.
2. Vykresli scénu normálním způsobem.
3. Převeď vrcholy do prostoru světla. A to `light_space_vertex = p_light * v_light * m * vertex`.
4. Uprav rozsah do rozsahu $< 0, 1 >$. A to `light_space_vertex = light_space_vertex * 0.5 + 0.5`. Následně použij jako texturovací souřadnice do stínové mapy.
5. Pro každý fragment proveď test, zda je pixel osvětlený nebo ne.

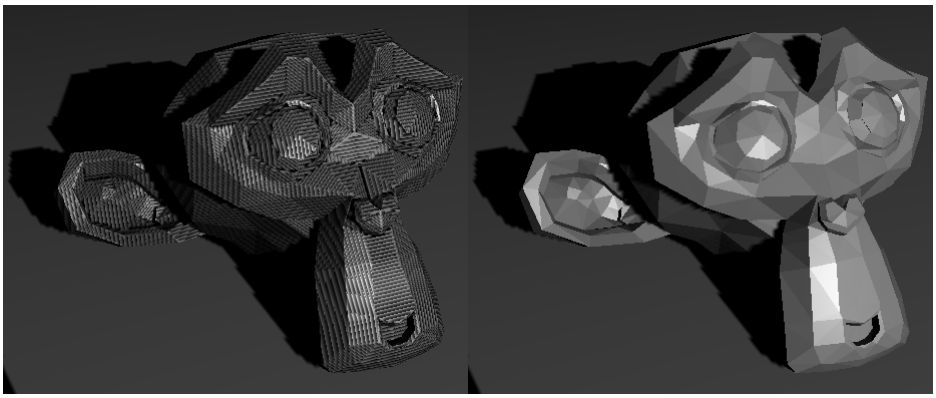
(a) Manuální porovnání (demonstrace viz obrázek č. 2.2). Pokud je vzdálenost Z_B bodu P od zdroje větší jak vzdálenost Z_A uložená ve stínové mapě, pak je fragment ve stínu, jinak je osvětlen.

(b) Automatické porovnání na hardwaru grafické karty a to nastavením parametru `GL_TEXTURE_COMPARE_MODE` a zvolením vhodnému typu sampleru v shaderu `sampler2DShadow`.

6. Vypočítej výslednou barvu fragmentu.

2.2.2 Aliasing stínů a stínové akné

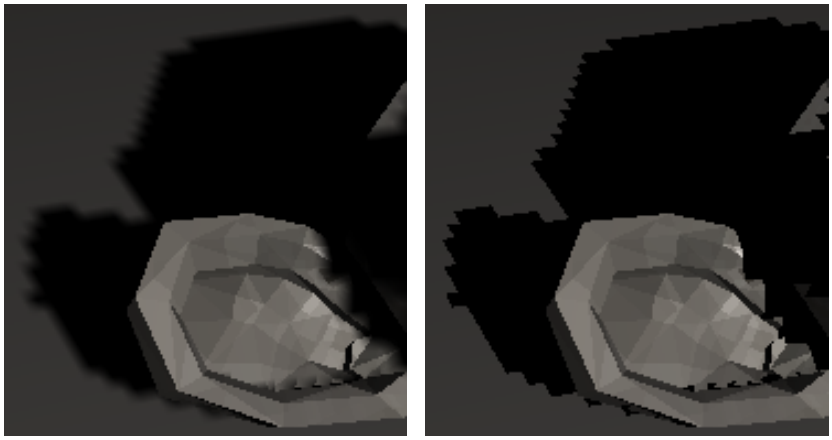
Jedním z nejčastějších problémů při použití *shadow mapping* algoritmu je vznik stínového akné (viz obrázek č. 2.3). Tento problém je způsobený omezenou přesností *z–bufferu*. A proto dochází k jevu, kdy se jeví hloubka pixelu shodná s hloubkou uloženou v hloubkové mapě. To způsobí, že daný pixel neprojde testem, zda je osvětlen, pak se může vykreslit na obrazovku jako zastíněný, i když má být osvětlený.



Obrázek 2.3: Chybný tzn. *self shadowing*. Vlevo scéna bez přidaného ϵ a vpravo po přidání ϵ.

Řešení existuje relativně mnoho, ale je nutné podotknout, že ani jedno nezabrání uplného vzniku akné. Pouze eliminují jeho vznik. Jednou z možností je zvýšit přesnost *z–bufferu*, dále je možné zvýšit rozlišení stínové mapy a nebo vykreslovat pouze odvrácené plochy geometrie do textury. Další možností je přičtení malého čísla k vypočtené hloubce fragmentu ϵ, kterému se říká *bias*. Problémem je, že pokud zvolíme příliš velké číslo, tak to způsobí, že se nám stíny posunou od objektu. Dále je vhodné vědět, že číslo ϵ se může pro každou scénu lišit a univerzální hodnota neexistuje.

Dalším velmi častým problémem je aliasing stínů. Je způsobený nízkým rozlišením stínové mapy a způsobuje zubaté hrany stínů (viz obrázek č. 2.4). Částečné řešení nabízí již *OpenGL* a to když nastavíme filtrovaní textury na `GL_LINEAR` (viz obrázek 2.4a). V této práci budeme dále pracovat s tímto filtrováním, jelikož se provádí hardwarově (při použití sampleru `sampler2Dshadow`) a částečně zlepšuje kvalita výsledných stínů. Tento problém lze dále vyřešit zvýšením rozlišení stínové mapy, což se negativně projeví na výkonu. Další možností je rozmazat hrany stínů nebo použití jedné mnoha technik stínových map jako např. perspektivní stínové mapy. V této práci jsem se měkkými stíny nezabýval.

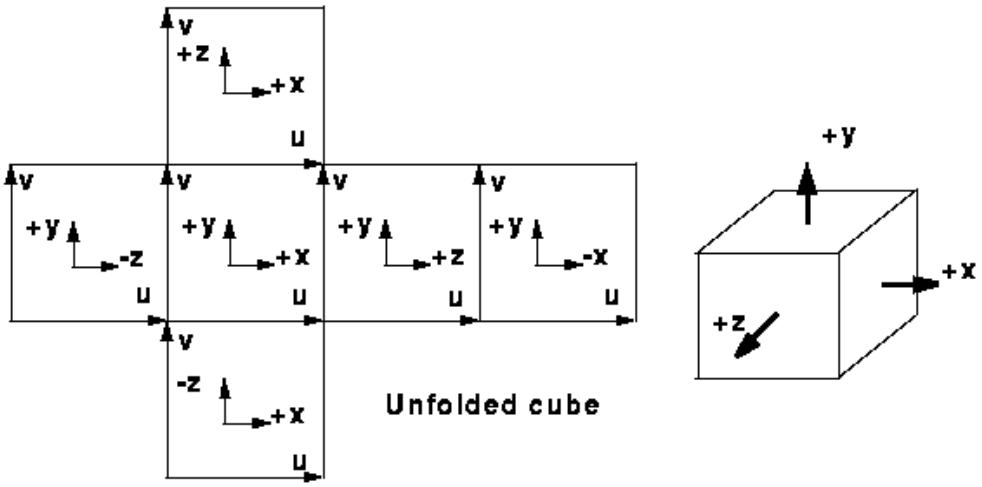


(a) `GL_LINEAR` (b) `GL_NEAREST`

Obrázek 2.4: Aliasing stínů při nízkém rozlišení stínové mapy. Porovnání různých nastavení filtrování textury.

2.3 Cube Map Shadow Mapping

Cube Map Shadow Mapping je modifikací předchozí metody. Upravuje ji pro použití se všesměrovými světlenými zdroji. Tato metoda vyžaduje místo jedné stínové mapy sadu 6 textur. Tuto sadu nazýváme *cube map* textura [1, 3]. V této textuře odpovídají jednotlivé textury stranám krychle (viz obrázek č. 2.5) a umožňuje pokrýt celé okolí zdroje světla, toho se mimo jiné používá také pro vykreslování odrazů okolí tzv. *environment mapping*.

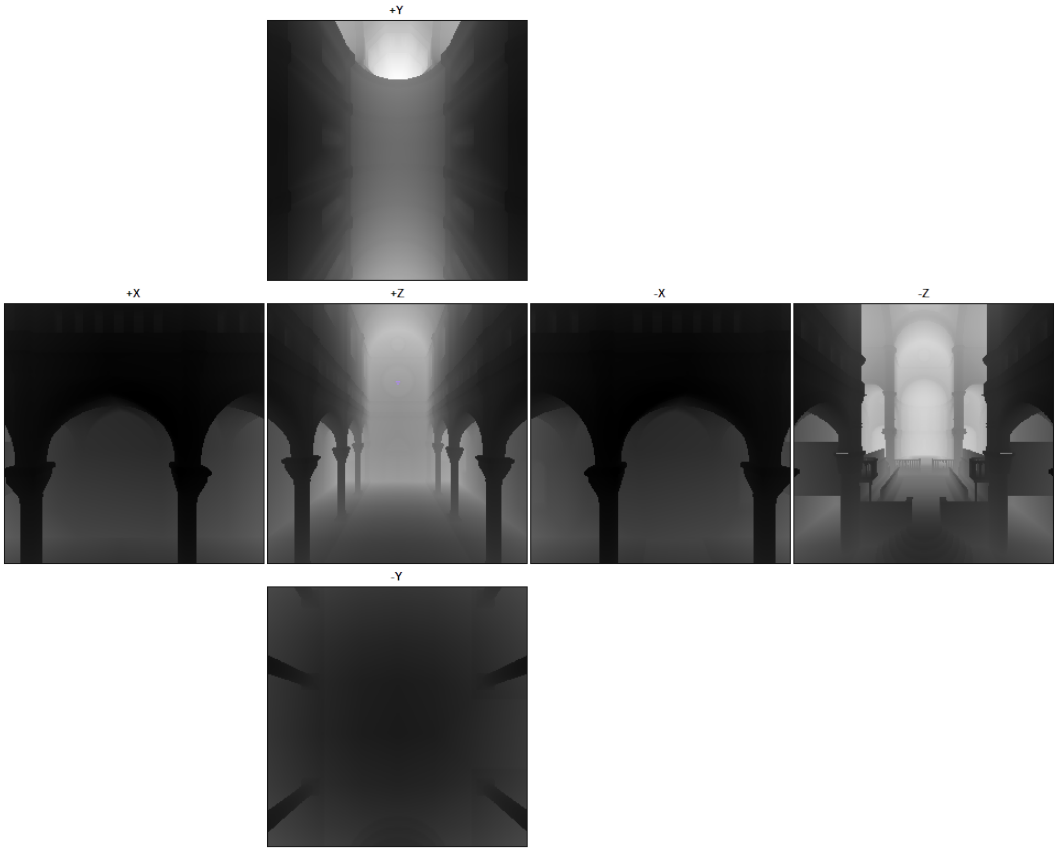


Obrázek 2.5: Rozbalená *cube map* textura [11].

Algoritmus je velice podobný původní metodě, ale je zde rozdíl v počtu průchodů. Kdy pro kompletní mapu je potřeba šest vykreslení hloubky scény z pozice světla. Každé vykreslení musí mít správnou orientaci. To znamená, že je třeba nejdříve vykreslit stranu `+X`, následně -X a tento postup opakovat dokud nemáme texturu kompletní. *Cube map* textura potřebuje pro namapování trojici souřadnic. Tyto souřadnice jsou (s, t, r) a představují

směrový vektor vycházející ze středu kostky. Podle komponenty tohoto vektoru, která je největší (má největší délku), se určí strana kostky. Následně se spočítají (s, t) souřadnice a provede normální namapování textury[1].

Nevýhodou této metody je, že bez optimalizace spotřebuje relativně dost času při vykreslování scény (nutnost 6 průchodů pro kompletní *cube map* texturu). I tento nedostatek se dá částečně vyřešit optimalizací metody. Mezi výhody patří to, že si poradí se všesměrovými světelnými zdroji a je relativně snadná na implementaci.



Obrázek 2.6: *cube map* textura s uloženou hloubkou scény. Obrázek pochází ze vzorové aplikace.

2.3.1 Algoritmus

1. Pro všechny strany *cube map* textury, vykresli scénu s vyplým osvětlením, stínováním a bez textur z pohledu světla a ulož její hloubku do patřičné textury.

2. Vykresli scénu normálním způsobem.

3. Vypočítej hloubku aktuálního fragmentu a směrový vektor r pro výběr textury. A to `r`

[1] Přesný mechanismus výběru strany a výpočtu texturovacích souřadnic je k nalezení ve specifikaci *OpenGL* na adrese: http://www.opengl.org/registry/specs/ARB/texture_cube_map.txt

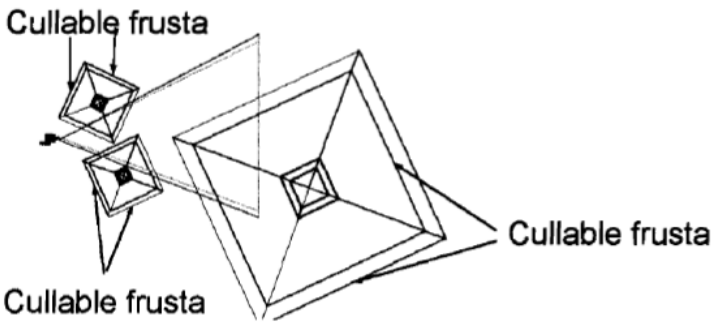
`= V_WorldSpacePosition.xyz - WorldSpace_lightPosition.xyz`. Výsledný vektor použij jako parametr pro texturovací funkci.

4. Pro každý fragment proveď test, zda je pixel osvětlený nebo ne. Buď manuálně nebo automaticky na kartě (vhodný sampler je `samplerCubeShadow`).

5. Vypočítej výslednou barvu fragmentu.

Optimalizace

Relativně lehce se dá dosáhnout toho, aby se v jistých případech nemusela scéna vykreslovat do všech šesti textur. A to v případě, když některé textury nejsou potřeba. Tím ušetříme čas potřebný pro naplnění stínové mapy, ale také ušetříme čas pro zpracování vrcholů. V praxi to znamená, pokud stojíme například zády ke světlu, tak není nutné vykreslovat do stínové mapy stěny krychle, které stejně nevidíme. Určení toho, co se bude nebo nebude vykreslovat, je založeno na průniku pohledových těles (viz obrázek 2.7) kamery a světelného zdroje pro aktuální stěnu *cube map* textury.



Obrázek 2.7: Průnik pohledových těles [8].

2.4 Dual Paraboloid Shadow Mapping

Další modifikace původního *shadow mapping* algoritmu. Tato metoda byla představena v článku *Shadow Mapping for Hemispherical and Omnidirectional Light Sources*[5] . Při použití této metody je scéna rozdělena na dvě polokoule. K pokrytí celé scény potřebujeme dva paraboloidy spojené zády k sobě, kdy každý paraboloid zachychuje jednu polokouli. Paraboloid je popsán rovnicí:

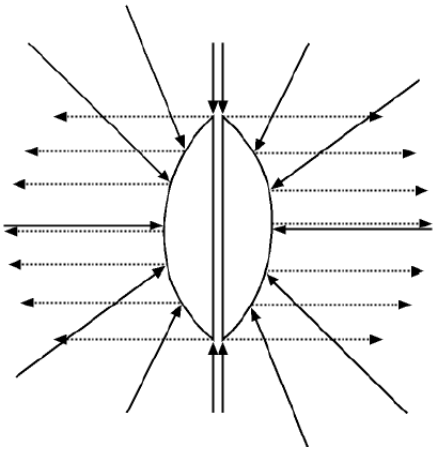
$$f(x,y) = \frac{1}{2} - \frac{1}{2}(x^2 + y^2), x^2 + y^2 \leq 1$$

Paraboloid si lze představit jako perfektně reflexní zrdcadlo, které odráží dopadající paprsky z polokoule do směru paraboloidu – jsou rovnoběžné se směrem paraboloidu (viz obrázek č. 2.8). Paprsky nesou informaci o vzdálenosti od světelného zdroje. Tuto informaci lze uložit do hloubkové textury (obrázek č.2.10). Principem je nalezení promítnutého bodu na povrch paraboloidu (viz obrázek č. 2.9).

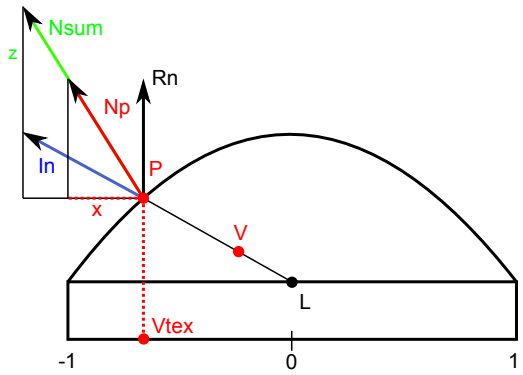
Musíme tedy najít x a y souřadnice promítnutého bodu. Tyto souřadnice lze spočítat za použití normálového vektoru v bodu projekce. Ten lze získat kartézským součinem tangent

(tečen) T_x a T_y, které získáme parciálními derivacemi podle x a y (viz rovnice 2.1 a 2.2).

$$\begin{aligned} P &= (x, y, f(x,y)) \\ T_x &= \frac{\partial P}{\partial x} = \left(1, 0, \frac{\partial f(x,y)}{\partial x}\right) = (1, 0, -x) \qquad (2.1) \\ T_y &= \frac{\partial P}{\partial x} = \left(0, 1, \frac{\partial f(x,y)}{\partial y}\right) = (0, 1, -y) \qquad (2.2) \\ N_P &= T_x \times T_y = (x, y, 1) \end{aligned}$$



Obrázek 2.8: Odraz dopadajících paprsků do směru paraboloidu [5].



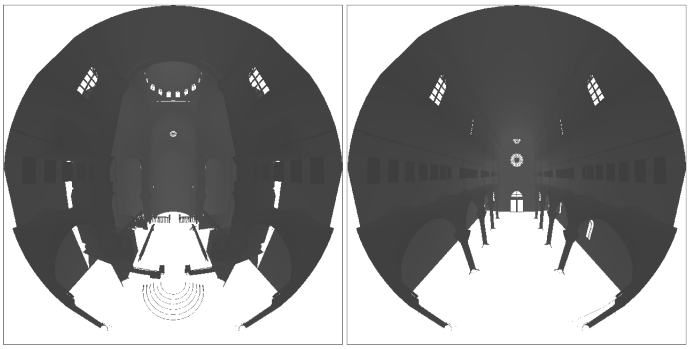
Obrázek 2.9: Demostrace hledání promítnutého bodu. Kde je `V` – bod (vrchol), `L` – světlo, `P` – průsečík, `Rn` – vektor odrazu paprsku, `In` – vektor dopadu paprsku, `Np` – normála kolmá na průsečík, `Nsum` – součet vektorů `In` a `Rn`, `Vtex` – promítnutý vrchol, `x` – výsledek. Převzato z *gamedevelop.eu*

Směr vektoru dopadajícího paprsku `In` odpovídá normalizované pozici vrcholu v parabolickém souřadném systému. Pokud k tomuto vektoru přičteme vektor odrazu `Rn`, dostaneme

vektor `Nsum`, který odpovídá normále v bodu projekce, pouze má jinou velikost. Jelikož vektor odrazu má stejný směr jako osa z, je nutné přičíst 1 k z–tové složce dopadajícího vektoru. Tyto znalosti nyní spojíme se vztahem, který využívá parciální derivace. A vyjde nám, že pokud vydělíme výsledný součet `Nsum` jeho z–tovou složkou, tak získáme x a y souřadnice hledaného promítnutého bodu (viz vztah 2.3).

$$\begin{aligned} N_P &= (x, y, 1) \Leftrightarrow I_n + R_n = N_{sum} \\ N_P &= \frac{N_{sum}}{z_{sum}} = \left(\frac{x_{sum}}{z_{sum}}, \frac{y_{sum}}{z_{sum}}, 1 \right) \end{aligned} \tag{2.3}$$

Výhodou této metody je především rychlost. Protože jsou potřeba pouze dva průchody pro zachycení celé scény. Toto s sebou nese ovšem také jistá omezení. Konkrétně je vyžadováno, aby objekty, které vrhají stín byly dostatečně tessellovány [12]. Nedostatečná tessellace má za následek nesprávné vrhání stínu nebo artefakty na místech, kde se oba paraboloidy spojují. Artefakty v místě spoje obou paraboloidů lze zredukovat vhodným zvolením umístění roviny, která rozděluje scénu na dvě polokoule. Problém s nesprávným vrháním stínu lze řešit několika způsoby. Je možné předpokládat, že má scéna dostatečně hustou síť vrcholů nebo je možné využít na nových grafických kartách možnosti hardwarové tessellace, která dokáže vrcholy dynamicky zahusťovat. V nejlepším případě by měla být tessellace dynamická tj. založená na vzdálenosti od světelného zdroje. V této práci byla implementována statická tessellace na GPU.

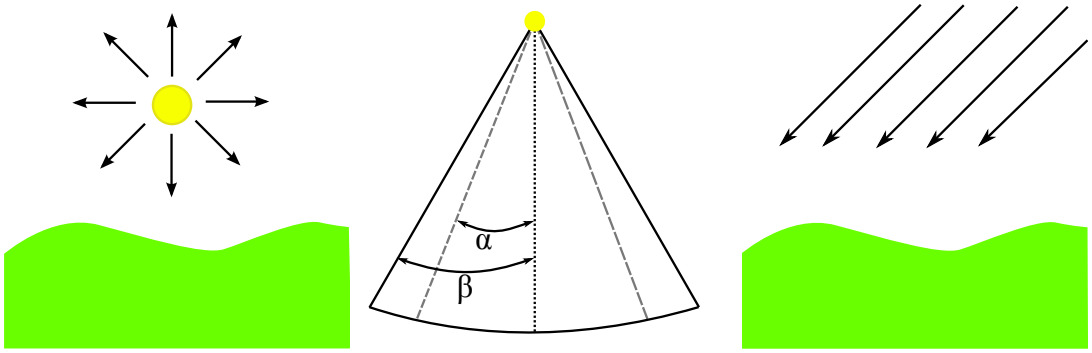


Obrázek 2.10: Hloubka scény uložená po parabolické projekci.

Kapitola 3

Světelné zdroje

Světelný zdroj v grafice je jeden z nejdůležitějších prvků. Umožňuje nám to, abychom vůbec něco viděli. V 3D grafice se vyskytuje více typů zdrojů světla. A v této kapitole si nastíníme několik typů.



(a) Bodový zdroj světla (b) Reflektor (c) Směrový zdroj světla

Obrázek 3.1: Zdroje světla

3.1 Směrové zdroje

Směrové světelné zdroje vyzařují světlo z nekonečné vzdálenosti a především pouze jedním definovaným směrem – parametr `d(x,y,z)` (obrázek č. 3.1c). Tímto světelným zdrojem se často modeluje například slunce. U tohoto typu zdroje so používá paralelní projekce.

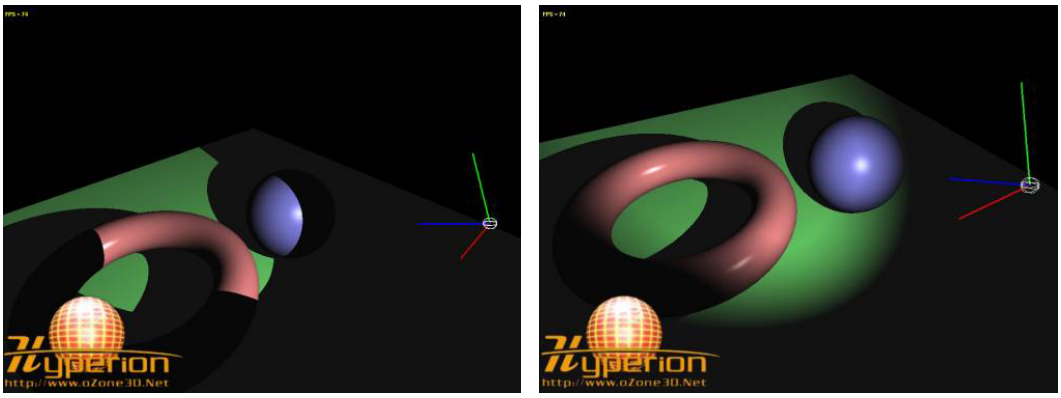
3.2 Všesměrové zdroje

Všesměrovému zdroji světla se také říká bodový. Z toho lze odvodit, jaký tvar tento světelný zdroj má. Je reprezentovaný nekonečně malým bodem – především jeho pozicí `p(x,y,z)`, který vyzařuje světlo rovnoměrně do všech směrů (obrázek č. 3.1a). Tento zdroj nemá v reálném světě konkrétní příklad, jako nejbližší příklad lze uvést žárovku.

3.3 Reflektor

Reflektor vychází ze všesměrového světelného zdroje. S tím rozdílem, že vyzařované světlo má tvar kuželu. V reálném světě si ho lze představit jako baterku. Jedná se o jeden ze nejpoužívanějších světelných zdrojů. Tento zdroj je reprezentován několika parametry. A to pozicí – `p(x,y,z)`, směrem vyzařování světla – `spotDir(x,y,z)` a dvojicí úhlů. Vnitřní – α a vnější – β (obrázek č. 3.1b). Tyto dva úhly se použijí pro porovnání, zda je oblast osvětlená nebo nikoliv. Čili pokud je $cos(\alpha)$ větší než $cos(\beta)$, je oblast v kuželu světla. Tento přístup vyprodukuje ovšem ostrou hranu na přechodu osvětlené a neosvětlené metody (viz obrázek 3.2a). Pokud bychom chtěli, aby tento přechod nebyl ostrý, ale postupný (viz obrázek 3.2b), potom lze použít parametru `falloff`. Tento parametr spočteme následovně.

```
falloff = clamp((cos_cur_angle - cos_outer_cone_angle) /
        cos_inner_minus_outer_angle, 0.0, 1.0);
```



(a) Bez *falloff* efektu (b) S *falloff* efektem

Obrázek 3.2: Porovnání *falloff* efektu u reflektoru. Převzato z *www.ozone3d.net*

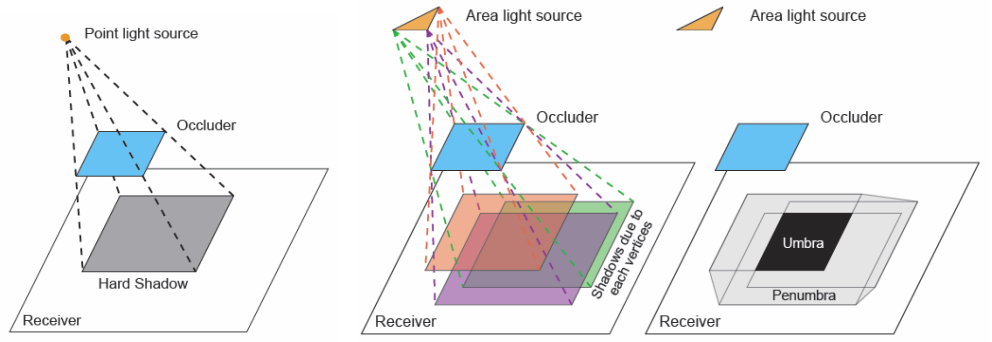
3.4 Plošné zdroje

Všechny objekty osvětlené předchozími zdroji světla vrhaly ostré stíny. Oproti tomu plošný světelný zdroj vrhá měkké stíny, tato skutečnost ovšem stojí výpočetní výkon. Je to dáno tím, že světlo vyzařuje z plochy oproti nekonečně malému bodu jako u předchozích zdrojů. Odtud dostal také tento zdroj své jméno.

3.5 Ostré a měkké stíny

Všechny výše uvedené světelné zdroje, kromě plošných zdrojů, produkují pouze ostré (*hard*) stíny (viz obrázek 3.3a). Takové stíny nejsou úplně reálné. V reálném světě se vyskytují stíny, které nemají ostrou hranu přechodu mezi oblastí ve stínu a osvětlenou oblastí. Takovým stínům v počítačové grafice říkáme měkké (*soft*) stíny (viz obrázek 3.3b). Pokud se budeme bavit o *shadow mapping* algoritmu, existuje několik technik [10, 4], které dokážou z tvrdých stínů udělat měkké.

Uvedeme si alespoň algoritmus *Percentage Closer Filtering*[14]. Princip tohoto algoritmu je vcelku jednoduchý. Místo toho, abychom četli ze stínové mapy jeden vzorek pro



(a) Geometrie tvrdého stínu (b) Geometrie měkkého stínu

Obrázek 3.3: Vznik ostrého a měkkého stínu [10].

určení, zda je pixel ve stínu, vezmeme v potaz okolí tohoto vzorku a hodnoty zprůměrujeme. Tímto dosáhneme toho, že se rozmažou hrany stínů a vytvoří dojem měkkého stínu. Okolí vzorku se říká jádro – *kernel*. Čím větší jádro, tím větší rozmazání okraje stínu. Nevyhodou této metody je ovšem náročnost na výkon. Pokud bychom chtěli měkké stíny a lepší výkon, je nutné sáhnout po lepších algoritmech jako například *Variance Shadow Maps*[7].

Kapitola 4

Demonstrační aplikace

Cílem práce je navrhnout demonstrační scénu a na ní otestovat vybrané stínové algoritmy. A to s použitím knihovny *OpenGL* a jazyka *GLSL*,

4.1 Návrh

Jelikož cílem práce je implementace stínových algoritmů a jejich následné otestování, tak by tomu měla odpovídat testovací scéna. Měla by být navržena tak, aby na ní bylo možné vidět silné i slabé stránky metody a to včetně případné optimalizace metody.

Dále by aplikace měla zobrazovat informace o aktuální použité metodě, času vykreslení do textury, rozlišení stínové mapy a případně o počtu nevykreslených stěn při zaplé optimalizaci. Mimo to je ještě vhodné zobrazovat aktuální pozici kamery ve scéně. Dále by aplikace měla umožnit za běhu přepínání obou implementovaných metod a také možnost zapnutí a vypnutí optimalizace. A v neposlední řadě by měla umožnit spustit animační průchod scénou nebo se přepnout do módu volné kamery.

4.2 Implementace

Aplikace byla napsána v jazyce C++ s využitím knihovny *OpenGL* verze 4.0. Ačkoli byla aplikace primárně vyvýjena pod operačním systémem *Microsoft Windows* neměl by být problém aplikaci přeložit a spustit pod jiným operačním systémem.

GLM

OpenGL Mathematics (GLM) je matematická knihovna distrubuovaná ve formě hlavičkového souboru, která je založená na specifikaci *OpenGL Shading Language (GLSL)*. Poskytuje třídy pro vektory, matice a funkce pro práci s nimi. Dále poskytuje funkce na vytvoření projekční matice, pohledové matice a mnoho dalšího. Výhodou je, že syntaxe je stejná jako při psaní shaderů.

Formát pro uložené modely

Aplikace načítá modely ve formátu *.obj*. Je to formát vyvinutý firmou *Wavefront Technologies* pro textové uložení geometrické reprezentace dat jako jsou vrcholy, normály, texturovací

souřadnice atp. Tento formát je vcelku jednoduchý a relativně snadný na načítání. Sám o sobě ovšem nenese informace o materiálu objektu, k tomu exituje další typ souboru a to *.mtl*. V současnosti je dost rozšířený a je podporovaný programy jako například Blender. Ve vzorové aplikaci je implementované jednoduché načítání modelů v tomto formátu, které nepracuje s materiály a pracuje pouze s trojúhelníkou sítí.

Model je načten do vnitřních struktur třídy `sceneObject` metodou `loadFile()`. Ukládá se pouze pozice bodu, normála a texturovací souřadnice. Následně se vytvoří indexy, které zajistí to, že každý bod bude poslán do grafické karty právě jednou.

Kamera

V rámci aplikace jsou implementovány dva typy kamery. Klasická FPS kamera, která se ovládá myší a klávesnicí a je reprezentována strukturou `ControlCamera`. U tohoto typu kamery bylo nutno vyřešit dva základní problémy. A to pohyb kamery a její natočení. Pohyb kamery vpřed, vzad nebo do stran je intuitivní. Stačí přičíst krok předem dané velikosti ke směrovému vektoru v případě pohybu vpřed, vzad nebo přičíst krok k *right* vektoru v případě pohybu do stran. Pro natočení pohledu kamery jsem využil funkcí goniemtrických funkcí *sinus* nebo *cosinus*. Pro výpočet souřadnic směrového vektoru kamery platí:

$$\begin{aligned} x &= r * sin(\varphi) * cos(\theta) \\ y &= r * sin(\varphi) * sin(\theta) \\ z &= r * cos(\varphi) \\ \overrightarrow{dir} &= (x, y, z) \end{aligned}$$

kde $r = 1$, $\varphi = \langle 0°, 360° \rangle$ a $\theta = \langle -90°, 90° \rangle$. Dále pro výpočet souřadnic vektoru *right* platí:

$$\overrightarrow{right} = (sin(\theta - 90°), 0, cos(\theta - 90°))$$

Následně stačí vypočítat vektor *up*, který je na tyto dva vektory kolmý:

$$\overrightarrow{up} = \overrightarrow{dir} \times \overrightarrow{right}$$

O výpočet projekční a pohledové matice se starají funkce `perspective()` a `lookAt()` z knihovny *GLM*.

Druhý typ kamery se využívá pro animaci a je reprezentován třídou `spline`. Tato kamera využívá kubický spline pro interpolaci mezi klíčovými snímky animace. Mezi dvěma klíčovými snímky je 200 interpolovaných pohledových matic. Ty vznikly z interpolovaných vektorů pozice kamery – *pos*, směru pohledu kamery – *dir* a směru vzhůru – *up*. Klíčové snímky byly pro scénu vybrány ručně s ohledem na to, aby se zde zohlednila i opřípadná optimalizace metody. A jsou uloženy v souboru `keyFrames.txt`.

Sestavení pohledového tělesa

Pohledové těleso je prostor, který je snímaný kamerou. V této práci budu pracovat s pohledovými tělesy, které vzniknou po použití perspektivní projekce. Toto výsledné těleso má tvar jehlanu s useknutou špičkou. Skládá se ze šesti stran a osmi vrcholů. Nicméně pro jeho sestrojení potřebujeme ještě pár bodů navíc (viz obrázek 4.1).

Nejdříve je nutné spočítat středy blízké a vzdálené ořezové roviny. Podle značení v obrázku 4.1 tedy *nearCenter* a *farCenter* a to:

$$\begin{aligned} nearCenter &= cameraPos + \overrightarrow{dir} * near \\ farCenter &= cameraPos + \overrightarrow{dir} * far \end{aligned}$$

Kde *camerPos* je pozice kamery ve scéně a *near* a *far* jsou vzdálenosti blízké a vzdálené ořezové roviny. Následně si spočítáme polovinu šířky a výšky vzdálené a blízké ořezové roviny.

$$\begin{aligned} nh &= near * tan(\varphi) \\ nw &= nh * aspectRatio \\ fh &= far * tan(\varphi) \\ fw &= fh * aspectRatio \end{aligned}$$

Kde φ je zorný úhel v radiánech a *aspectRatio* je poměr stran vzniklý vydělením šírky a výšky zobrazovaného okna. Následně si spočítáme rohy pohledového tělesa pro zadní ořezovou rovinu.

$$\begin{aligned} ftl &= farCenter + (\overrightarrow{up} * fh) - (\overrightarrow{right} * fw) \\ ftr &= farCenter + (\overrightarrow{up} * fh) + (\overrightarrow{right} * fw) \\ fbl &= farCenter - (\overrightarrow{up} * fh) - (\overrightarrow{right} * fw) \\ fbr &= farCenter - (\overrightarrow{up} * fh) + (\overrightarrow{right} * fw) \end{aligned}$$

A pro blízkou ořezovou rovinu.

$$\begin{aligned} ntl &= nearCenter + (\overrightarrow{up} * nh) - (\overrightarrow{right} * nw) \\ ntr &= nearCenter + (\overrightarrow{up} * nh) + (\overrightarrow{right} * nw) \\ nbl &= nearCenter - (\overrightarrow{up} * nh) - (\overrightarrow{right} * nw) \\ nbr &= nearCenter - (\overrightarrow{up} * nh) + (\overrightarrow{right} * nw) \end{aligned}$$

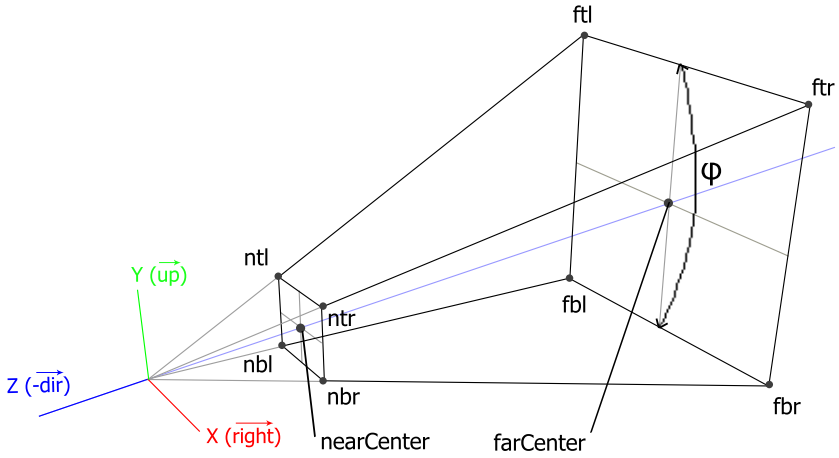
A nakonec si spočítáme roviny pohledového tělesa. Ty jsou definovány třemi body, které byly spočteny výše.

$$\begin{aligned} TOP &: ntr, ntl, ftl \\ BOTTOM &: nbl, nbr, fbr \\ LEFT &: ntl, nbl, fbl \\ RIGHT &: nbr, ntr, fbr \\ NEAR\_PLANE &: ntl, ntr, nbr \\ FAR\_PLANE &: ftr, ftl, fbl \end{aligned}$$

Ve výsledné aplikaci má výpočet na starosti funkce `createFrustumPoints()`.

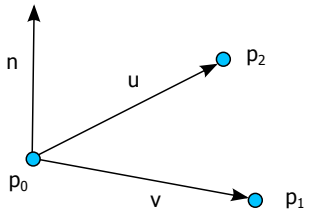
Průnik dvou pohledových těles

Test průniku dvou pohledových těles je rozdělen na dva symetrické testy. V každém testu se nejdříve zjistí, zda leží nějaký z rohů prvního pohledového tělesa uvnitř druhého pohledového tělesa. Pokud ano, znamená to, že se obě tělesa protínají a není nutné provádět



Obrázek 4.1: Pohledové těleso s vyznačenými body pro jeho sestrojení.

následující test. Další test zjišťuje, zda nějaká hrana prvního pohledového tělesa neprotíná jednu z rovin druhého pohledového tělesa. Pokud neprojde ani tento test, pak lze říct, že se tělesa neprotínají. Algoritmus je popsán v knize *ShaderX*^{3} *Advanced Rendering with DirectX and OpenGL* [8]. Tyto testy jsou implementovány ve funkci `intersection()`.



Obrázek 4.2: Vytvoření roviny pomocí trojice bodů. Převzato z *lighthouse3d.com*

Testování bod–rovina a přímka–rovina

Zda bod leží v pohledovém tělese, lze určit tak, že si spočítáme vzdálenost bodu od rovin pohledového tělesa. Pokud je tato vzdálenost záporná, tak bod leží mimo pohledové těléso, jinak leží uvnitř. To ovšem platí za předpokladu, že normály rovin pohledového tělesa směřují dovnitř tělesa. Pokud budeme vycházet z toho, že rovinu lze zapsat následovně:

$$P : Ax + By + Cz + D = 0 \tag{4.1}$$

Pak lze vzdálenost bodu $p(p_x, p_y, p_z)$ od roviny spočítat následovně:

$$distance = A * p_x + B * p_y + C * p_z + D = n \cdot p + D$$

Kde n je normála roviny. Test, zda leží bod v pohledovém tělesu, je implementován ve funkci `pointInfrustum()` a vzdálenost počítá v aplikaci funkce `distancePointPlane()`.

Této vzdálenosti lze využít i při testu, zda přímka protíná nějakou z rovin. Pokud je přímka dána body p_1 a p_2, pak stačí spočítat vzdálenost těchto bodů od roviny. Pokud jsou obě vzdálenosti záporné, tak přímka leží mimo, jinak přímka protíná pohledové těleso. Tento test obstarává funkce `edgeIntersectFrustum()`.

Samotnou rovinu popsanou rovnicí 4.1 lze vytvořit pomocí trojice bodů p_0, p_1, p_2 [9]. Toto je znázorněno na obrázku 4.2. Funkce `createPlaneFromPoints()` se ve výsledné aplikaci postará o vytvoření roviny.

Vykreslení textu

Jelikož jsem se rozhodl, že ve výsledné aplikaci budu na obrazovku vypisovat text, musel jsem vyřešit jeho zobrazení. Bohužel samotná knihovna *OpenGL* nemá žádnou funkci na vykreslení textu. Z toho důvodu jsem se rozhodl použít externí knihovnu *SDL_ttf*, která umožňuje vykreslení `TrueType` fontů na obrazovku. O vykreslení se stará třída `textRenderer`. Při vykresování se postupuje následovně:

1. Inicializace *SDL_ttf* pomocí funkce `TTF_Init()` a následné zvolení fontu funkcí `TTF_OpenFont()`.
2. O samotné vykreslení textu se postará funkce `TTF_RenderText_Blended()`.
3. Následně se vytvoří textura a zkopíruje se do ní výsledek z předchozího kroku.
4. Pro správné zobrazení je nutné povolit `glEnable(GL_BLEND)` a nastavit `glBlendFunc(GL_SRC_ALPHA, GL_ONE_MINUS_SRC_ALPHA)`.

Framebuffer objekty

Framebuffer jako takový je obrazové výstupní zařízení, které přenáší obrazovou informaci z bufferů a obsahuje právě jeden vykreslený snímek. *Framebuffer object* je objekt, který umožňuje vytvoření vlastních uživatelem definovaných framebufferů. Díky nim lze vykreslovat mimo základní framebuffer a to tak aniž bychom měnili obsah na obrazovace. Tohoto lze s úspěchem využít v situaci, kdy potřebujeme z nějakého důvodu (například za účelem vytvoření stínové mapy) vykreslit scénu nebo její část do textury. Obvyklý postup je následující (příklad z výsledné aplikace pro *cube map* texturu):

1. Vygenerování framebuffer objektu – `glGenFramebuffers(1, &CubeFBOs[i]);`
2. Nastavení (nabindování) framebuffer objektu – `glBindFramebuffer(GL_FRAMEBUFFER, CubeFBOs[i]);`
3. Nastavení cíle vykreslování (připojení textury) a parametrů vykreslení (např. jen hloubková komponenta) – `glFramebufferTexture2D(GL_FRAMEBUFFER, GL_DEPTH_ATTACHMENT, GL_TEXTURE_CUBE_MAP_POSITIVE_X + i, CubeTextureShadow, 0);`
4. Vykreslení scény do framebuffer objektu
5. Odnastavení framebuffer objektu – `glBindFramebuffer(GL_FRAMEBUFFER, 0);`

V aplikaci se o nastavení framebuffer objektů stará funkce `initFBO()`. Připojení více textur (např. *cube* mapa) je v aplikaci řešeno vytvořením pole framebuffer objektů.

Měření času vykreslení snímku

Jelikož cílem práce je také otestování implementovaných algoritmů, tak je nutné vyřešit, jak změřit čas strávený různými výpočty. V práci jsem zaznamenával čas potřebný pro naplnění stínové mapy. První řešení je použití časovače operačního systému (například funkcí `SDL_getTicks()`), pak výsledný čas je jednoduchým rozdílem časů na začátku vykreslovací smyčky a konci vykreslovací smyčky. Problémem tohoto řešení je fakt, že změřený čas by nemusel být správný. Jelikož se měří čas strávený na procesoru a *OpenGL* provádí nějaké výpočty na pozadí, tak systémový časovač vrátí hodnotu rovnou nebo blízkou nule.

Druhým řešením je použití časovače v knihovně *OpenGL*, který umožňuje zjistit, kolik času zabralo vykonání příkazu na grafické kartě. Jedná se o rozšíření *OpenGL Timer Queries* a je dostupné ve verzi *OpenGL 3.3* a novější. Při měření se postupuje následovně:

1. Začít dotazování (*query*) na výpočetní čas – `glBeginQuery(GL_TIME_ELAPSED, queries[1]);`
2. Provést vykreslovací operace
3. Zastavit dotazování na výpočetní čas – `glEndQuery(GL_TIME_ELAPSED);`
4. Zažádat *OpenGL* o výsledek dotazu – `glGetQueryObjectui64v(queries[1], GL_QUERY_RESULT, &queryResults[1]);`

Problém může nastat, když se pokusíme získat výsledek a ten není ještě připravený. V tom případě musí *OpenGL* počkat než se provedou měřené operace na pozadí, čož může mít za následek snížení výkonu.

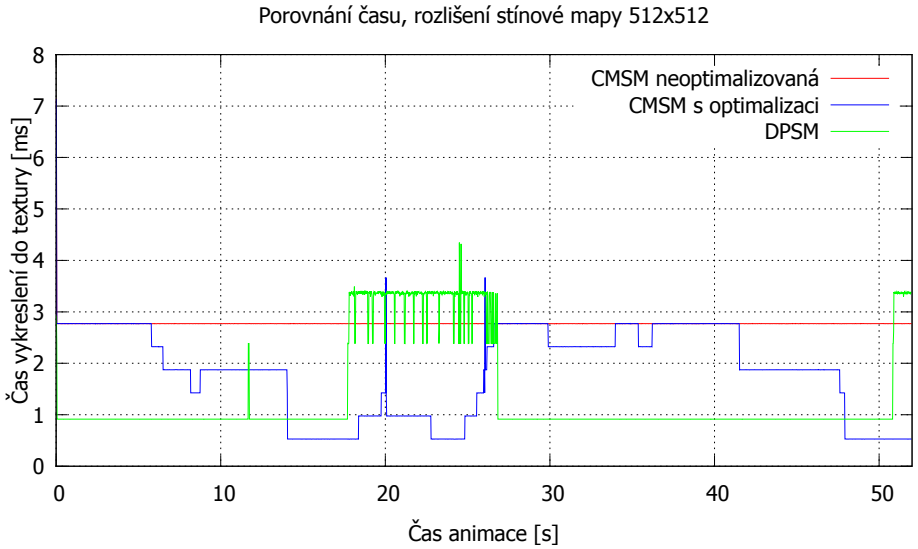
Kapitola 5

Srovnání metod

V této kapitole se zaměříme na otestování implementovaných metod. Zejména na časy potřebné pro vykreslení scény do textury. Dále se podíváme na kvalitativní srovnání a na to, jak je jaká metoda náročná na video paměť. U kvalitativního srovnání provedeme přepočet paměti potřebné pro uložení textury. Tím docílíme objektivnějšího srovnání kvality obou metod.

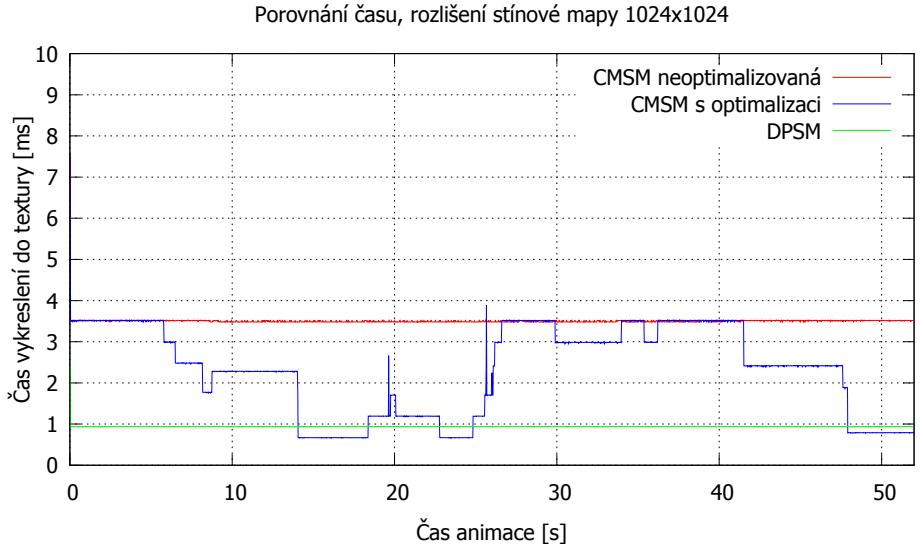
5.1 Čas vykreslení

Jednou z velmi důležitých věcí, které mají vliv na celkový výkon je právě čas, který grafická karta spotřebuje pro vytvoření stínové mapy. Tento čas by měl být co nejmenší. Na následujících grafech je vyneseno, kolik v jaký okamžik potřebovala grafická karta času pro vykreslení do textury. Tyto časy pochází z animačního průletu kamery scénou, který trval 52 vteřin.



Obrázek 5.1: Porovnání časů potřebných pro uložení scény do textury pro rozlišení stínové mapy 512∗512 pixelů.

Z grafu 5.1 lze vyčíst, že neoptimalizovaná varianta 6–průchodového algoritmu si drží téměř konstantní dobu vykreslení. To se ovšem nedá říct o metodě využívající parabolickou projekci. Ta má několik zákmitů, kde je dokonce čas vykreslení do textury větší než u neoptimalizované varianty 6–průchodového algoritmu. Ačkoli by teoreticky i tato metoda měla mít téměř konstantní dobu vykreslení, tak i po provedení série testů se zákmity stále vyskytovali. Tento problém ovšem nenastává při zvýšení rozlišení stínové mapy, jak je možné vidět na grafech 5.2, 5.3 a 5.4.



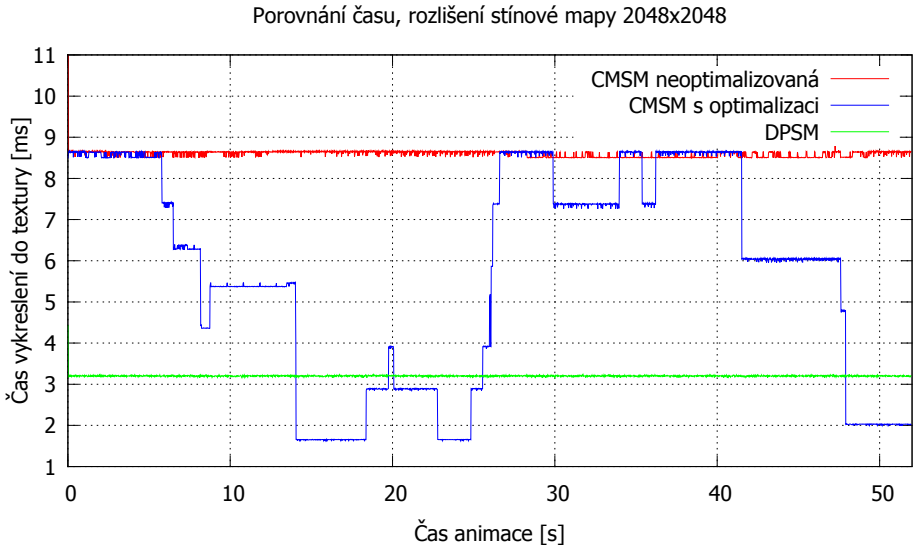
Obrázek 5.2: Porovnání časů potřebných pro uložení scény do textury pro rozlišení stínové mapy 1024∗1024 pixelů.

V grafech 5.2, 5.3 a 5.4 lze vidět, že doby vykreslení dosahují do textury teoreticky předpokládaných hodnot. Za povšimnutí stojí metoda využívající parabolickou projekci. Ta si drží relativně nízké časy vykreslení i při vysokém rozlišení stínové mapy. A při pohledu do tabulky 5.1, kde jsou uvedeny průměrné časy pro vykreslení do textury, vidíme, že například při rozlišení stínové mapy 4096∗4096 pixelů je průměrný čas 6.766061 ms. Což je přibližně třikrát méně než průměrný čas optimalizovaného 6–průchodového algoritmu a přibližně čtyřikrát méně než u neoptimalizované metody.

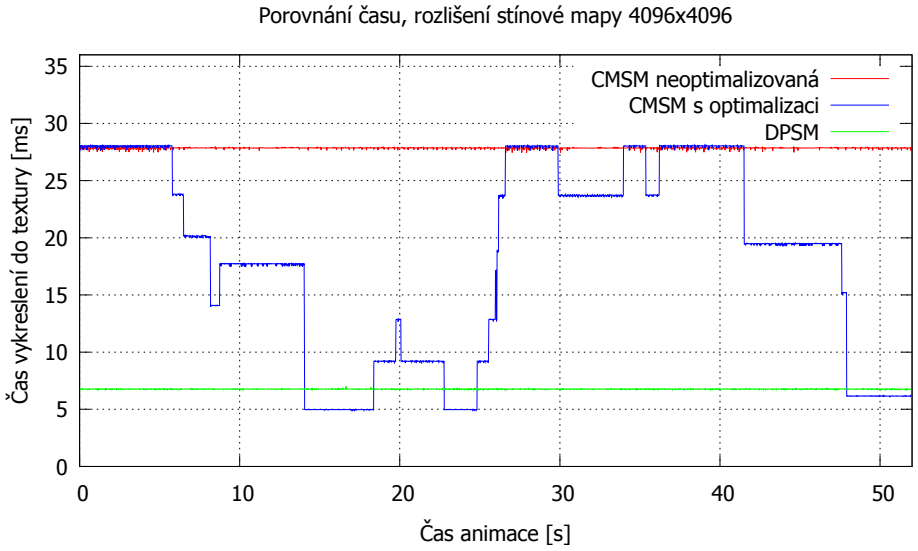
V grafu 5.5 lze vidět, jak se mění počet vykreslených stěn kubické mapy při použití optimalizace.

Rozlišení stínové mapy [px]	**DPSM [ms]**	**CMSM [ms]**	**CMSM optim. [ms]**
512*512	1.376003	2.77259	1.834874
1024*1024	0.938634	3.50093	2.221608
2048*2048	3.201865	8.598977	5.716675
4096*4096	6.766061	27.84751	18.4383

Tabulka 5.1: Průměrné časy vykreslení do textury.



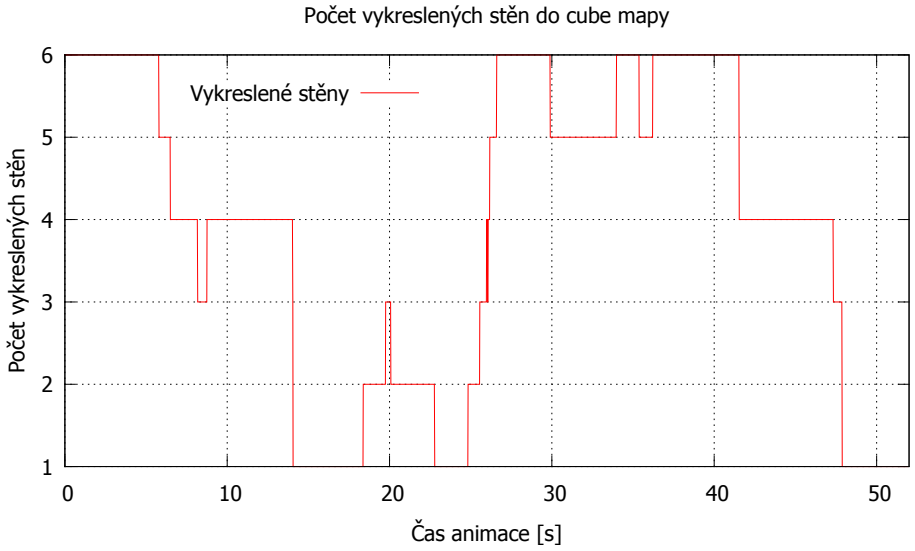
Obrázek 5.3: Porovnání časů potřebných pro uložení scény do textury pro rozlišení stínové mapy 2048∗2048 pixelů.



Obrázek 5.4: Porovnání časů potřebných pro uložení scény do textury pro rozlišení stínové mapy 4096∗4096 pixelů.

5.2 Kvalitativní porovnání

Porovnávat obě metody, když mají stejné rozlišení textur, není příliš objektivní. Jelikož metoda *Cube map shadow mapping* má k dispozici přibližně třikrát více paměti, takže by měla dávat podstatně kvalitnější výsledky. Kolik přibližně paměti spotřebuje daná metoda,



Obrázek 5.5: Graf počtu vykreslených stěn kubické mapy.

lze celkem jednoduše spočítat a to podle vztahu:

$$m = x * \frac{sm\_res^2 * depth\_bits}{8 * 1000} \tag{5.1}$$

Kde, m je spotřebovaná pamět v `kB`, x je počet textur (např. u parabolické metody se $x = 2$), sm_res je rozlišení textury a $depth\_bits$ určuje přesnost v bitech, v aplikaci je $depth\_bits = 32$. V tabulce 5.2 je vidět paměťová náročnost metod při stejném rozlišení textur.

Rozlišení stínové mapy [px]	**Spočtená pamět [kB]**		**Reálná paměť [kB]**	
	DPSM	CMSM	DPSM	CMSM
512*512	2097	6291	2048	6144
1024*1024	8388	25165	8192	24576
2048*2048	33554	100663	32768	98304
4096*4096	134208	402624	131072	393216

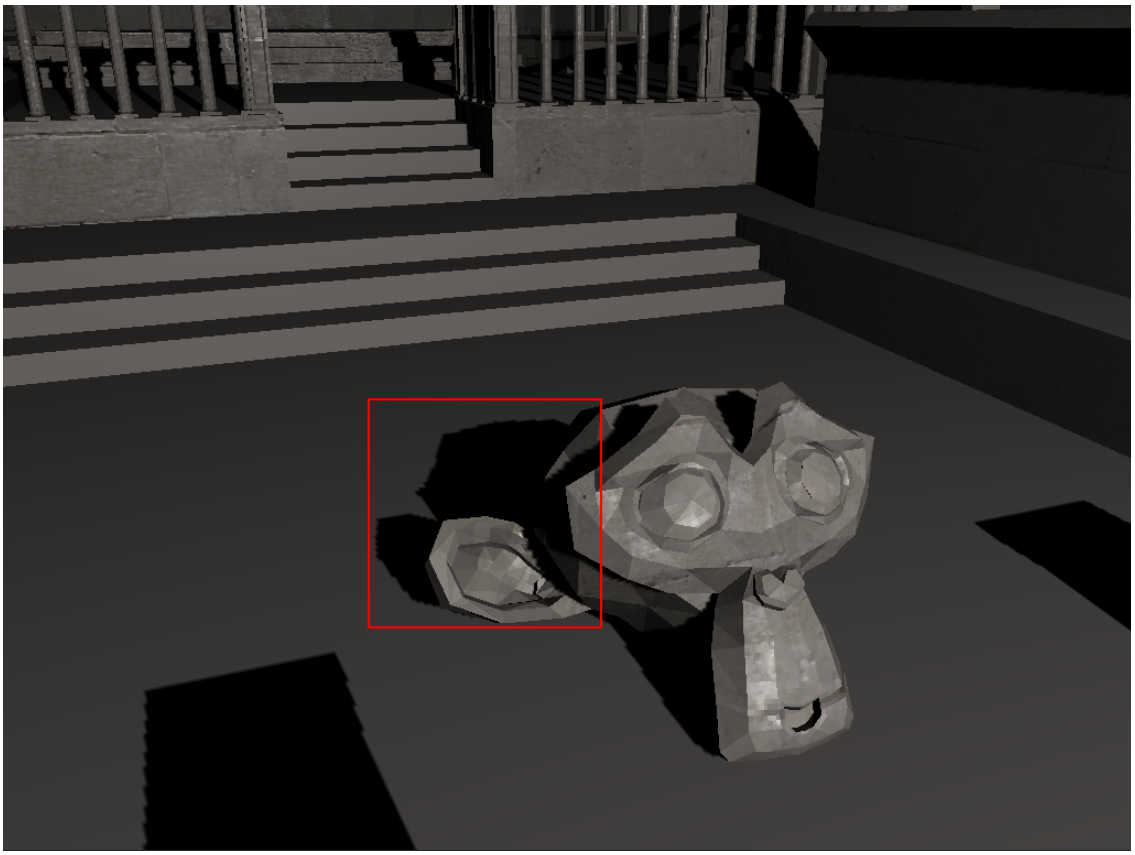
Tabulka 5.2: Množství použité paměti u metod. Hodnoty ve sloupci "Reálná pamět" jsou převzaty z aplikace *gDEBugger*.

Relativně objektivnější srovnání obou metod bude v případě, že obě metody budou mít k dispozici stejné množství paměti. Přepočteme množství použité paměti u *Cube map shadow mapping* tak, aby přibližně odpovídalo množství použité paměti parabolické projekci. Pokud vyjdeme ze vztahu 5.1, lze vyjádřením členu sm_res (viz vztah 5.2) spočítat potřebné rozlišení textury. V tabulce 5.3 jsou uvedené přepočtené hodnoty rozlišení stínové mapy.

$$sm\_res = \sqrt{\frac{8 * 1000 * m}{x * depth\_bits}} \tag{5.2}$$

Metoda	Rozlišení stínové mapy [px]		
DPSM	512*512	1024*1024	2048*2048
CMSM	295*295	591*591	1182*1182

Tabulka 5.3: Přepočtené rozlišení textury.

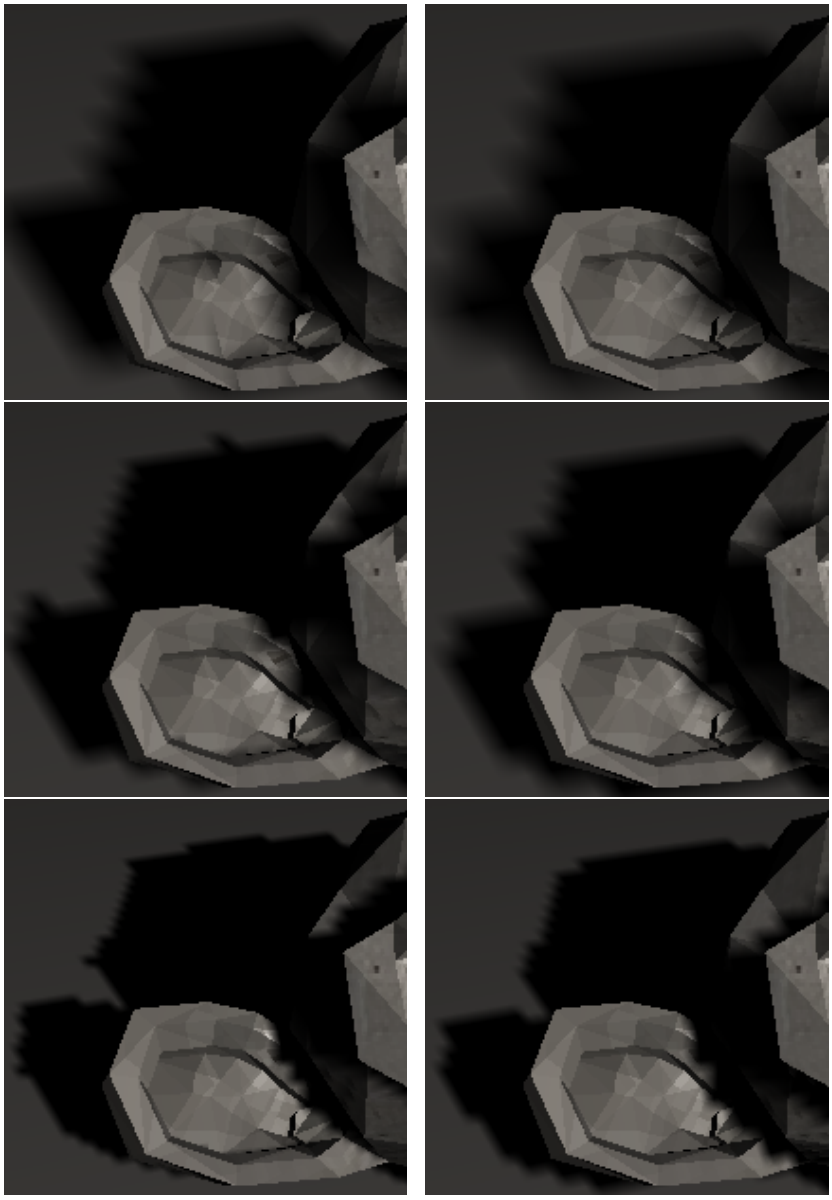


Obrázek 5.6: Referenční obrázek pro kvalitativní porovnání. Oblast označená v rámečku bude sloužit jako testovací vzorek.

Pro vizuální porovnání kvality budeme vycházet z obrázku 5.6. Pokud se detailněji podíváme na zvětšenou oblast (viz obrázek č. 5.7) původního obrázku, tak se ukáže, že při použití stejného množství paměti se kvalita obou metod vcelku srovnala. Ale i přesto je podle mého názoru kvalita metody *Cube map shadow mapping* o trochu lepší. Zdá se, že se zachycuje trochu detailněji geometrii. To může být způsobeno i tím, že při parabolické projekci se čte informace pouze z jedné stínové mapy na polovinu scény. Ve vzorové scéně druhá metoda sice také čte z jedné stínové mapy, ale s tím rozdílem, že tato mapa pokrývá mnohem menší plochu a může tak detailněji obsáhnout zobrazovanou geometrii. A to i napříč tomu, že má podstatně menší rozlišení stínové mapy.

Ovšem druhou stranou je otázka teoretického výkonu. Zda stojí za nepatrné zvýšení kvality za snížení výkonu, pokud se bavíme o neoptimalizované metodě *Cube map shadow mapping*. U malých rozlišení je výkon metody pracující s kubickou mapou ještě relativně přijatelný, ale s vyšším rozlišením rapidně klesá. S její optimalizací se situace zlepší, ale v

nejhorším případě na tom bude hůře než metoda parabolické projekce.



(a) CMSM (b) DPSM

Obrázek 5.7: Detail na vybranou oblast. Rozlišení hloubkové mapy je podle tabulky 5.3.

Kapitola 6

Závěr

Cílem práce bylo seznámit se s problematikou vytváření stínů ve scéně za použití knihovny *OpenGL* a jazyka *GLSL*. V kapitole 2 jsou popsány základní algoritmy pro zobrazení stínů. Dále byly nastudovány algoritmy *Cube map shadow mapping* a *Dual paraboloid mapping*, které vycházejí ze základního *Shadow mapping* algoritmu a rozšiřují ho o podporu všesměrových světelných zdrojů. V rámci algoritmu *Cube map shadow mapping* byla implementována optimalizační metoda, která snižuje dobu vykreslení. Pro tyto algoritmy byla navržena scéna, na které byly demonstrovány a otestovány. V rámci testování popsaného v kapitole 5 bylo provedeno srovnání nejen výkonu algoritmů, ale také výsledné kvality stínů.

Výsledky testů ukázaly, že metoda používající parabolickou projekci je podle předpokladu několikrát rychlejší než neoptimalizovaná metoda používající kubickou mapu. Pokud budeme vycházet z tabulky 5.1, potom se toto zrychlení pohybuje v řádu 100–300% v závislosti na rozlišení stínové mapy. Dále testy ukázaly, že použitá optimalizace metody *Cube map shadow mapping* přinesla zrychlení přibližně 50% oproti neoptimalizované variantě, avšak je stále přibližně o 30–180% pomalejší než při použití parabolické projekce.

Při porovnání kvality výsledných stínů záleží na tom, jak se při samotném porovnání postupuje. Při stejném rozlišení stínových map poskytuje metoda *Cube map shadow mapping* kvalitnější výsledky. Ovšem pokud přidělíme stejné množství paměti pro uložení textur oběma metodám, tak je výsledná kvalita srovnatelná. U parabolické projekce se vyskytuje artefakt na spoji obou paraboloidů, který je nejspíše způsobený pevným umístěním dělící roviny. Tento problém ovšem zmizí, pokud se do stínové mapy vykreslí pouze odvrácené strany objektů.

Možností jak rozšířit tuto práci je hned několik. V první řadě by bylo vhodné rozšířit algoritmy tak, aby produkovali měkké stíny. Například o metodu *Percentage Closer Filtering*. Dalším možným rozšířením by mohla být podpora pohyblivého zdroje světla nebo podpora více světelných zdrojů. Také lze práci rozšířit o implementaci algoritmu stínových těles nebo rozšířit o další optimalizace pro již použité metody.

Literatura

[1] *OpenGL Cube Map Texturing* [online]. 1999 [cit. 3. února 2013]. Dostupné na: <http://www.nvidia.com/object/cube_map_ogl_tutorial.html>.

[2] *Chapter 9. Advanced Topics* [online]. 2003 [cit. 3. února 2013]. Dostupné na: <http://http.developer.nvidia.com/CgTutorial/cg_tutorial_chapter09.html>.

[3] *Cubemap Texture* [online]. 2013 [cit. 3. února 2013]. Dostupné na: <http://www.opengl.org/wiki/Cubemap_Texture>.

[4] Agrawala, M., Ramamoorthi, R., Heirich, A. et al. Efficient image-based methods for rendering soft shadows. In *Proceedings of the 27th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM Press/Addison-Wesley Publishing Co., 2000. S. 375–384. SIGGRAPH '00. Dostupné na: <http://dx.doi.org/10.1145/344779.344954>. ISBN 1-58113-208-5.

[5] Brabec, S., Annen, T. a Seidel, H. Shadow mapping for hemispherical and omnidirectional light sources. In *Proc. of Computer Graphics International*. 2002. S. 397–408.

[6] Crow, F. C. Shadow algorithms for computer graphics. In *Proceedings of the 4th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1977. S. 242–248. SIGGRAPH '77. Dostupné na: <http://doi.acm.org/10.1145/563858.563901>.

[7] Donnelly, W. a Lauritzen, A. Variance shadow maps. In *Proceedings of the 2006 symposium on Interactive 3D graphics and games*. New York, NY, USA: ACM, 2006. S. 161–165. I3D '06. Dostupné na: <http://doi.acm.org/10.1145/1111411.1111440>. ISBN 1-59593-295-X.

[8] Engel, W. *Shaderx 3: Advanced Rendering With DirectX And OpenGL*. [b.m.]: Charles River Media, 2005. ISBN 1-58450-357-2.

[9] Ericson, C. *Real-Time Collision Detection (The Morgan Kaufmann Series in Interactive 3-D Technology) (The Morgan Kaufmann Series in Interactive 3D Technology)*. San Francisco, CA, USA: Morgan Kaufmann Publishers Inc., 2004. ISBN 1558607323.

[10] Hasenfratz, J.-M., Lapierre, M., Holzschuch, N. et al. A survey of Real-Time Soft Shadows Algorithms. *Computer Graphics Forum*. Dec 2003, roč. 22, č. 4. S. 753–774. Dostupné na: <http://maverick.inria.fr/Publications/2003/HLHS03a>.

[11] Mizutani, Y. a Reindel, K. *Environment Mapping Algorithms* [online]. [cit. 3. února 2013]. Dostupné na: <http://www.reindelsoftware.com/Documents/Mapping/Mapping.html>.

[12] Osman, B., Bukowski, M. a McEvoy, C. Practical implementation of dual paraboloid shadow maps. In *Proceedings of the 2006 ACM SIGGRAPH symposium on Videogames*. New York, NY, USA: ACM, 2006. S. 103–106. Sandbox '06. Dostupné na: <http://doi.acm.org/10.1145/1183316.1183331>. ISBN 1-59593-386-7.

[13] Žára, J., Beneš, B., Sochor, J. et al. *Moderní počítačová grafika*. 2. Praha: Computer Press, 2005. ISBN 80-251-0454-0.

[14] Reeves, W. T., Salesin, D. H. a Cook, R. L. Rendering antialiased shadows with depth maps. In *Proceedings of the 14th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1987. S. 283–291. SIGGRAPH '87. Dostupné na: <http://doi.acm.org/10.1145/37401.37435>. ISBN 0-89791-227-6.

[15] Williams, L. Casting curved shadows on curved surfaces. In *Proceedings of the 5th annual conference on Computer graphics and interactive techniques*. New York, NY, USA: ACM, 1978. S. 270–274. SIGGRAPH '78. Dostupné na: <http://doi.acm.org/10.1145/800248.807402>.

Příloha A

Obsah CD

Přiložený datový nosič obsahuje následující věci:

- Adresář `bin` – obsahuje spustitelnou aplikaci pod operačním systémem *Microsoft Windows* včetně `.dll` knihoven
- Adresář `src` – obsahuje zdrojové kódy, projekt a solution pro *Microsoft Visual Studio*, dále obsahuje všechny potřebné knihovny a hlavičkové soubory
- Adresář `text_src` – obsahuje zdrojové kódy textové části práce pro systém LaTeX
- `projekt.pdf` – tento dokument v elektronické verzi
- `README.txt` – stručný popis ovládání aplikace a nastavení projektu pro *Microsoft Visual Studio*